```python
import torch
import torchvision as tv


class AE( torch.nn.Module):
    def __init__( _, vsize, hsize):
        super().__init__()
        _.enc = torch.nn.Sequential( torch.nn.Linear( vsize, hsize),
                                     torch.nn.Tanh())
        _.dec = torch.nn.Sequential( torch.nn.Linear( hsize, vsize),
                                     torch.nn.Tanh())

    def forward( _, x):
        return _.dec( _.enc( x))        # enc y dec se usan como funciones.

# N, M, P = 10, 3, 1000
# m = torch.randn( M, N)
# z = torch.randn( P, M)
# x = torch.mm( z, m).tanh()
# ae = AE( N, M)                        # El AE puede decorrelacionar vars.

# torch.save( ae.state_dict(), "ae.state")      # Se puede grabar...
# ae.load_state_dict( torch.load( "ae.state"))  # ...y volver a recuperar.


class SAE( torch.nn.Module):            # Stacked Auto-Encoders.
    def __init__( _, sizes):
        super().__init__()
        _.subnet = torch.nn.ModuleList()
        for i in range(len(sizes)-1):
            _.subnet.append( AE(sizes[i],sizes[i+1]))

    def enc( _, x, depth=None):
        depth = len(_.subnet) if depth is None else depth+1
        xi = x
        for i in range(depth):
            xi = _.subnet[i].enc( xi)
        return xi

    def dec( _, y, depth=None):
        depth = len(_.subnet) if depth is None else depth+1
        yi = y
        for i in reversed(range(depth)):
            yi = _.subnet[i].dec( yi)
        return yi

    def forward( _, x, depth=None):     # Varia la profundidad del AE.
        yi = _.enc(  x, depth)
        xi = _.dec( yi, depth)
        return xi, yi
```

```python
T = 20
B = 50
N = 28*28
M = 64
C = 10

transf = tv.transforms.Compose(
    [ tv.transforms.ToTensor(),
      tv.transforms.Normalize([0.5],[0.5]) ])   # Normalizar por la Tanh.

trn_data = tv.datasets.MNIST( root='./data', train=True,
                              download=True, transform=transf )

# ...mas tst_data, trn_load y tst_load.

sizes = [ N, 512, 128, M]
model = SAE( sizes)
optim = torch.optim.Adam( model.parameters())   # Le podemos pasar todos.
costf = torch.nn.MSELoss()

model.train()
for depth in range(len(model.subnet)):
    print( "Depth:", depth)               # Pero si hiciera falta elegir...
    #optim = torch.optim.Adam( model.subnet[:depth+1].parameters())
    for t in range(T):
        E = 0
        for images, labels in trn_load:
            optim.zero_grad()
            data = images.view( -1, N)
            x, y = model( data, depth)
            loss = costf( x, data)
            loss.backward()
            optim.step()
            E += loss.item()
        print( t, E)

lincl = torch.nn.Linear( M, C)
optim = torch.optim.Adam( lincl.parameters())
costf = torch.nn.CrossEntropyLoss()
for t in range(T):
    E = 0
    for images, labels in trn_load:
        optim.zero_grad()
        x = images.view( -1, N)
        y = model.enc( x)
        cp = lincl( y)
        error = costf( cp, labels)
        error.backward()
        optim.step()
        E += error.item()
    print( t, E)

# Acurracy como siempre.
```